Efficient Edge-Swapping Heuristics for the Reload Cost Spanning Tree Problem

S. Raghavan and Mustafa Sahin
The Robert H. Smith School of Business and Institute for Systems Research, University of Maryland, College Park, Maryland 20742, USA

The reload cost spanning tree problem (RCSTP) is an NP-hard problem, where we are given a set of nonnegative pairwise demands between nodes, each edge is colored and a reload cost is incurred when a color change occurs on the path between a pair of demand nodes. The goal is to find a spanning tree with minimum total reload cost. We propose a tree–nontree edge swap neighborhood for the RCSTP and an efficient way to search this neighborhood using preprocessed information. We then embed this edge swap neighborhood within a local search and a tabu search heuristic. We also discuss an initial solution procedure that is used by the local search and tabu search heuristic in a multistart framework. On a test set of 630 instances (that includes benchmark instances from Gamvros et al. [6]), the local search solution improves upon the initial solution in 416 instances by an average of 23.62%, and the tabu search solution improves upon the local search solution in 364 instances by an average of 35.79%. Out of 495 test instances from this set that we know the optimal solutions for, the initial solution is optimal 113 times, the local search solution is optimal 224 times, and the tabu search solution is optimal 481 times. On a second set of benchmark instances from Khalil and Singh [9], the tabu search solution improves upon the best known solution in 32 out of 44 instances. **© 2015 Wiley Periodicals, Inc. NETWORKS, Vol. 65(4), 380–394 2015**

Keywords: edge swap; local search; network optimization; reload cost; spanning tree; tabu search; heuristic; neighborhood search

1. INTRODUCTION

The minimum reload cost spanning tree problem (RCSTP) is an NP-hard combinatorial optimization problem introduced by Gamvros et al. [6]. Mathematically, the RCSTP is defined on an undirected graph $G = (V, E)$ where we are given a color c_{ij} for each edge $(i, j) \in E$, and a set of nonnegative demands d_{st} between all pairs of nodes (s, t) in V. For each node i in V and for all pairs of its incident edges $((i, j), (i, k))$, let $r^i(c_{ij}, c_{ik})$ be the per unit demand reload cost based on the colors of the edges (i, j) and (i, k). Given a spanning tree T, the total reload cost between nodes s and t is the sum of per unit demand reload costs incurred by the color changes along the unique path from s to t in T multiplied by the demand d_{st}. The total reload cost of a spanning tree T is given by the sum of reload costs of all pairs of nodes. The objective is to build a spanning tree T over the nodes in V with the minimum total reload cost.

There are many real-life settings where the concept of reload occurs naturally. For instance, in telecommunication networks, reload costs are incurred when data is transferred between diverse technologies. In intermodal freight transportation, unloading and loading the freight from one type of carrier to the next results in a reload cost. In the energy industry, the loss of energy during its transfer between carriers can be captured by introducing reload costs to the network. Another important occurrence of reloads is in the context of disaster relief. Here, links may collapse or connection points/intersections may be blocked by obstacles, which necessitates additional reload costs in the transportation network used to provide disaster relief. To elaborate, suppose a bridge over a body of water collapses. Then another mode of transport must be used to transport relief supplies across the body of water. In this situation, a reload cost is occurred on each end of the bridge, as supplies carried to one endpoint of the bridge must be transferred into a ship incurring a reload cost, then transferred from the ship back to a mode of land transport at the other endpoint of the bridge, incurring another reload cost. On the other hand if a disaster results in a blocked intersection, an additional reload cost is incurred for supplies moved through that intersection. This corresponds to the cost of physically unloading and loading supplies from one transportation vehicle to another at that intersection.

Even though many real-life settings have reload costs, the concept of reload did not receive much attention from the research community up until the last decade. The first article to feature reload costs was by Wirth and Steffan [11] where they introduce the minimum diameter spanning tree with reload costs. The problem is to build a spanning tree

Received August 2013; accepted February 2015
Correspondence to: S. Raghavan; e-mail: raghavan@rhsmith.umd.edu
DOI 10.1002/net.21609
Published online 20 March 2015 in Wiley Online Library (wileyonlinelibrary.com).

that has the smallest diameter with respect to the reload costs. Subsequent to its introduction, Galbiati [4] solves an open problem left by Wirth and Steffan [11] by showing unless P=NP, the problem of finding a minimum diameter spanning tree for a graph with maximum degree of 4, cannot be approximated within any constant $\alpha < 2$ if the reload costs are unrestricted, and cannot be approximated within any constant $\alpha < 5/3$ if the reload costs satisfy the triangle inequality. Amaldi et al. [2] study the complexity and approximability of the problems of finding optimal paths, tours, and flows under a cost model including reloads and regular costs. In Galbiati et al. [5], the authors consider the minimum reload cost cycle cover problem and prove that it is strongly NP-hard and not approximable within any $\varepsilon > 0$ even when the number of colors is 2, reload costs are symmetric and satisfy the triangle inequality. Gourvès et al. [8] study the complexity of the minimum reload cost *s*-*t* path, trail, and walk problems with reload costs. They consider both symmetric and asymmetric reload costs and situations with general costs and where the triangle inequality is satisfied. In addition to proving complexity results, they identify some special cases that are polynomially solvable. Gamvros et al. [6] show the RCSTP is NP-hard, provide two mixed integer linear programming formulations for the problem and show that both models have the same linear programming relaxation bounds. They introduce several variants of the RCSTP and perform extensive computational experiments on the different formulations, which we will use as benchmark instances in this article.

Given the natural occurrence of reloads in many applications and the fact that the problem is NP-hard, there is a need for heuristics that produce high-quality solutions for reload cost problems. Yet, most of the work has been done to produce exact solution approaches and prove theoretical bounds on the approximability of the problem. To the best of our knowledge, the only heuristic approach on the problem to date is by Khalil and Singh [9] where the authors apply an ant colony optimization heuristic. The main goal of this paper is to propose a high-quality heuristic and validate its quality and efficiency by evaluating it on a large set of test instances (including some benchmark instances from the literature). We propose a local search neighborhood for the RCSTP that involves a tree–nontree edge swap and an efficient way to search this neighborhood using preprocessed information. This enables us to effectively insert this local search neighborhood within a tabu search framework and rapidly find high-quality solutions to large RCSTP instances.

In the literature, the tree–nontree edge swap neighborhood has been used previously—especially for NP-hard spanning tree problems. Ahuja and Murty [1] use the tree–nontree edge swap neighborhood in a local search algorithm for the optimal communication spanning tree problem (OCSTP).[1] There are some similarities and significant differences between Ahuja and Murty's use of this tree–nontree edge swap neighborhood which we will elaborate on later in this paper. Amaldi et al. [3] use the tree–nontree edge swap neighborhood in a local search, a tabu search, and a variable neighborhood search algorithm in order to find good quality solutions to the minimum fundamental cycle basis problem. Silva et al. [10] use the tree–nontree edge swap neighborhood in a local search heuristic for the spanning tree problem with the minimum number of branch vertices. Here the goal is to design a spanning tree with the fewest number of vertices with degree greater than 2. In contrast to the previous three papers where the tree–nontree edge swap is used to find an improved solution (within a heuristic for an NP-hard problem), there are combinatorial optimization problems where the goal is to find the best tree–nontree edge swap. For example, Wu et al. [12] discuss a problem where there are k source nodes and weights on each edge. Given a spanning tree, the total routing cost is defined as the sum of distances from all nodes to sources. In this setting, a tree edge can undergo a transient failure, in which case it needs to be replaced by a nontree edge. The objective is then to find the best nontree edge such that the total routing cost is minimized. The goal is to find for each tree edge the best nontree edge to swap it with in the case of a transient failure. Notice that the problem is polynomially solvable and the relevant papers are focused on finding the fastest algorithms to compute these best tree–nontree edge swaps. Let $|V| = n$ and $|E| = m$. Wu et al. propose an $\mathcal{O}(m \log n + n^2)$ algorithm for the case of two sources and an $\mathcal{O}(mn)$ algorithm for the case of more than two sources. Gfeller [7] studies tree–nontree edge swaps for the minimum diameter spanning tree, in a similar fashion to [12]. Here for a given tree edge (that can undergo a transient failure), we wish to replace it by the best nontree edge that minimizes the diameter of the spanning tree. The goal is to find all of the best swaps for the edges on the tree. Again the problem is polynomially solvable. Gfeller improves the running time to $\mathcal{O}(m \log n)$ using $\mathcal{O}(m)$ space.

The rest of this paper is organized as follows. Section 2 describes the tree–nontree edge swap neighborhood. Section 2.1 discusses how to efficiently search through the neighborhood and identify the best edge swap. Section 2.2 explains how to update the relevant preprocessed information after an edge swap is performed. Section 3 presents an initial solution procedure, as well as the local search and tabu search algorithms based on the edge swap neighborhood. The local search and tabu search algorithms are run in a multistart framework. In section 4, we consider the single-source RCSTP and the single-source fixed RCSTP and adapt the heuristics for these two variants. Section 5 reports on our computational experience. Section 6 provides concluding remarks. In the Appendix, we discuss how to obtain a slight improvement in the running time of a single local search iteration in Ahuja and Murty's [1] algorithm.

[1] In the OCSTP, we are given nonnegative demands d_{st} between all pairs of nodes (s, t) on a graph $G = (V, E)$. The cost of any edge is simply its cost multiplied by the demand flowing on it. The objective is to find a spanning tree of minimum cost.

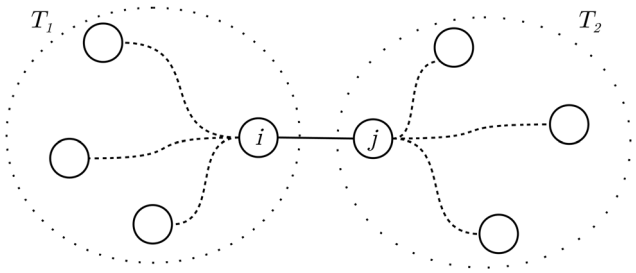

FIG. 1. Removing e_{ij} partitions T into two subtrees T_1 and T_2.

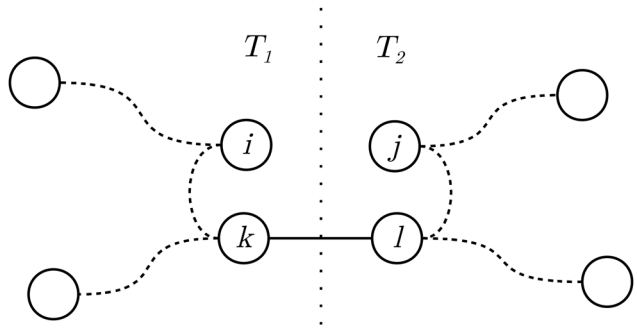

FIG. 2. After swapping e_{ij} and e_{kl} the partitions T_1 and T_2 remain unchanged. Note that it is possible to have $i = k$ or $j = l$ and the observation is still valid.

2. EDGE-SWAPPING NEIGHBORHOOD FOR THE RCSTP

Given a spanning tree T, we consider the tree–nontree edge swap neighborhood of T. For all edges $e_{ij} \in T$, we evaluate all possible edge swaps with edges $e_{kl} \in E \backslash T$ such that $T \backslash \{e_{ij}\} \cup \{e_{kl}\}$ is also a spanning tree, then we select the best edge swap. While this neighborhood is simple to state, care needs to be taken in implementing it. In order to effectively search through it and build a high-quality heuristic, it is very important to use any preprocessed information that would allow us to exploit the properties of spanning trees as it will greatly affect the efficacy and speed of the search. To that end, we divide a local search iteration into two parts. First, we identify the best edge swap efficiently by using preprocessed information. Then we update the tree and the associated information based on the edge swap performed.

2.1. *Identifying the Best Edge Swap*

For any edge $e_{ij} \in T$, as depicted in Figure 1, the removal of e_{ij} will result in two disjoint spanning trees T_1 and T_2 such that $T = T_1 \cup T_2 \cup \{e_{ij}\}$ and $i \in T_1, j \in T_2$. Let $R(T)$ denote the total reload cost of tree T. Then $R(T)$ can be broken down into two components. The first component is the total reload cost of the subtrees T_1 and T_2, denoted by $R(T_1)$ and $R(T_2)$, respectively (i.e., the total reload cost associated with demands whose origin and destination are both in T_1 or both in T_2). The second component is the total reload cost of the demand going from T_1 to T_2 and from T_2 to T_1. We will denote by $R(T, T_1 \rightarrow T_2)$ the total reload cost of the entire demand from T_1 to T_2 carried in T. Thus, the total reload cost of the spanning T can be stated as

$$R(T) = R(T_1) + R(T_2) + R(T, T_1 \rightarrow T_2) + R(T, T_2 \rightarrow T_1).$$

In order to calculate $R(T, T_1 \rightarrow T_2)$, we exploit the fact that for a given node $s \in T_1$, the entire demand $\sum_{t \in T_2} d_{st}$ has to follow the unique path from node s to node j in T before it enters T_2. For notational convenience, $P(s, j)$ denotes the path from node s to j. Similarly, for a given node $t \in T_2$, the entire demand $\sum_{s \in T_1} d_{st}$ has to follow the path $P(i, t)$ in T after it leaves T_1. We can think of e_{ij} as a bridge between T_1 and T_2, and any demand between T_1 and T_2 has to pass the bridge e_{ij}, so we have

$$R(T, T_1 \rightarrow T_2) = \sum_{s \in T_1} R(T, P(s,j)) \sum_{t \in T_2} d_{st} + \sum_{t \in T_2} R(T, P(i,t)) \sum_{s \in T_1} d_{st},$$

where $R(T, P(s,t))$ denotes the sum of unit reload costs on a given path $P(s, t)$ in T. Note that since the choice of T_1 and T_2 is arbitrary, we only provide definitions for one subtree and omit the definitions for the other (i.e., $R(T, T_2 \rightarrow T_1)$ is analogously defined).

Let T' be the spanning tree we obtain after making the edge swap $e_{ij} \in T$ and $e_{kl} \in E \backslash T$ as in Figure 2. We wish to calculate the cost difference between T and T' efficiently. Since T and T' differ by exactly one edge, $T \backslash T' = \{e_{ij}\}, T' \backslash T = \{e_{kl}\}$, the removal of e_{ij} from T and the removal of e_{kl} from T' would create the same two disjoint spanning trees T_1 and T_2. Without loss of generality, suppose $k \in T_1, l \in T_2$. Let $\delta(T, T') = R(T') - R(T)$ denote the difference in the total reload cost between T and T'. The subtrees T_1 and T_2 are the same for trees T and T' (see Fig. 2), therefore the total reload cost of T' can be expressed as $R(T') = R(T_1) + R(T_2) + R(T', T_1 \rightarrow T_2) + R(T', T_2 \rightarrow T_1)$ where

$$R(T', T_1 \rightarrow T_2) = \sum_{s \in T_1} R(T', P(s,l)) \sum_{t \in T_2} d_{st} + \sum_{t \in T_2} R(T', P(k,t)) \sum_{s \in T_1} d_{st}.$$

Notice that the reload costs for demands whose origin and destination stay within the subtree (i.e., within T_1 and T_2) are not affected by the swap. Thus, the difference in cost is dependent upon how the total reload cost associated with the demands from T_1 to T_2 and T_2 to T_1 change. $\delta(T, T') = [R(T', T_1 \rightarrow T_2) - R(T, T_1 \rightarrow T_2)] + [R(T', T_2 \rightarrow T_1) - R(T, T_2 \rightarrow T_1)]$. Even though any path $P(s, t)$ $s, t \in T_1$, $P(s, t)$ $s, t \in T_2$ and their respective unit reload costs $R(T, P(s,t))$ $s, t \in T_1, R(T, P(s,t))$ $s, t \in T_2$ stay exactly the same after the edge swap, paths $P(s, t)$ $s \in T_1, t \in T_2$, and $P(s, t)$ $s \in T_2, t \in T_1$ are subject to

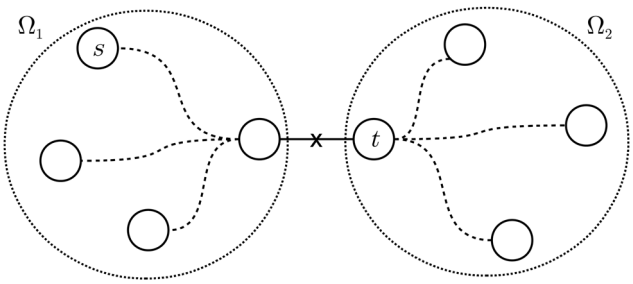

FIG. 3. Removing $e_{pred(s,t),t}$ partitions T into two subtrees Ω_1 and Ω_2.

change. Any path $P(s,t)$ from T such that $s \in T_1, t \in T_2$,

$$P(s,t) = P(s,i) \cup \{e_{ij}\} \cup P(j,t),$$

becomes,

$$P(s,t) = P(s,k) \cup \{e_{kl}\} \cup P(l,t),$$

in T' now that it has to cross the bridge e_{kl} instead of e_{ij}. Thus, we have

$$R(T', P(s,t)) = R(T, P(s,k)) + r^k(c_{\text{pred}(s,k),k}, c_{kl}) + r^l(c_{kl}, c_{l,\text{pred}(t,l)}) + R(T, P(l,t)), s \in T_1, t \in T_2,$$

where pred(s,t) denotes the predecessor node of t on the path from s to t in the tree T.

Consider a pair of nodes s and t in T as in Figure 3. Removing edge $e_{\text{pred}(s,t),t}$ creates two subtrees Ω_1 and Ω_2 such that $s \in \Omega_1$ and $t \in \Omega_2$. Let, q_{st}^- denote the total demand **from** node s to the nodes in Ω_2. Similarly, let q_{st}^+ denote the total demand **to** node s from the nodes in Ω_2.

With this notation, we can state $\delta(T,T')$ as

$$\begin{aligned}\delta(T,T') = &\sum_{t\in T_2}[R(T',P(k,t)) - R(T,P(i,t))]q_{ti}^+ \\ &+ \sum_{t\in T_1}[R(T',P(l,t)) - R(T,P(j,t))]q_{tj}^+ \\ &+ \sum_{s\in T_1}[R(T',P(s,l)) - R(T,P(s,j))]q_{sj}^- \\ &+ \sum_{s\in T_2}[R(T',P(s,k)) - R(T,P(s,i))]q_{si}^-.\end{aligned}$$

Recall $m = |E|$ and $n = |V|$. Then there are exactly $n-1$ edges in any given spanning tree T, which leaves $m-n+1$ edges that are not in the spanning tree. Therefore, for any spanning tree T, we have $\mathcal{O}((n-1)(m-n+1)) = \mathcal{O}(mn)$ different spanning trees T' that can be obtained by replacing a tree edge with a nontree edge. Suppose we store $R(T,P(s,t)), q_{st}^-, q_{st}^+$ for all $s,t \in T$ in matrices $\boldsymbol{R_p}$ and $\boldsymbol{Q^-}, \boldsymbol{Q^+}$,[2] then we could compute $\delta(T,T')$ in $\mathcal{O}(n)$ time since we would have $\mathcal{O}(1)$ time access to $R(T',P(s,t))$ via $R(T,P(s,t))$ for all $s,t \in T$. In a local search iteration, we start from a spanning tree T and iterate to another tree T' by comparing each of the $\mathcal{O}(mn)$ edge swaps and select the edge swap that minimizes $\delta(T,T')$. The iteration requires $\mathcal{O}(mn^2)$ time to complete in the worst case. After implementing an edge swap, we need to update the $\boldsymbol{R_p}, \boldsymbol{Q^-}, \boldsymbol{Q^+}$ matrices and predecessors to ensure that we can calculate the cost of an edge swap in the next local search iteration as efficiently.

[2] In other words, we store the reload cost of all paths in T and q_{st}^- and q_{st}^+ for all pairs of nodes in T.

2.2. *Updating the Preprocessed Information*

Suppose we accepted the edge swap $e_{ij} \in T$ with $e_{kl} \in E \backslash T$ such that $i,k \in T_1$ and $j,l \in T_2$ changing the current spanning tree from T to T', then we only need to update the entries of $\boldsymbol{R_p}$ for $s \in T_1, t \in T_2$ and $s \in T_2, t \in T_1$ since the edge swap only affects the paths between T_1 and T_2. Therefore, we replace the entries $R(T,P(s,t))$ with $R(T',P(s,t))$ for $s \in T_1, t \in T_2$ as in Algorithm 1; the algorithm for updating $R(T,P(s,t))$ for $s \in T_2, t \in T_1$ is omitted since it follows a similar procedure. In the worst case, there are $n/2$ nodes in both T_1 and T_2, which makes the update of $\boldsymbol{R_p}$ run in $\mathcal{O}(n^2)$ time.

Algorithm 1. Reload Cost Update

```
1: Input: T, T_1, T_2, e_kl
2: for all s ∈ T_1 do
3:    for all t ∈ T_2 do
4:       R(T,P(s,t)) = R(T,P(s,k)) + r^k(c_pred(s,k),k, c_kl) +
         r^l(c_kl, c_l,pred(t,l)) + R(T,P(l,t))
5:    end for
6: end for
```

However, we need to keep predecessors to update $\boldsymbol{R_p}$ which means we also need to update predecessors after the edge swap so that we can access them in $\mathcal{O}(1)$ time in the next iteration. Adding a nontree edge e_{kl} to T creates a cycle; let φ_1 denote the list of nodes that are in the cycle and in T_1 and let φ_2 denote the list of nodes that are in the cycle and in T_2. Note that $i,k \in \varphi_1$ and $j,l \in \varphi_2$. Without loss of generality, assume the nodes in φ_2 are ordered and indexed in the cycle direction from node l and ending in node j. After the edge swap, we only need to update the predecessors pred(s,t) such that $s \in T_1, t \in \varphi_2$ and $s \in T_2, t \in \varphi_1$. The other predecessors remain unaffected by the edge swap. We employ the procedure stated in Algorithm 2 for the update which is explained in Figure 4. Note that we only provide the update of predecessors for the paths originating in T_1 since the process is identical for the paths originating in T_2. There are $\mathcal{O}(n)$ nodes in any given cycle, so updating predecessors from T to T' is $\mathcal{O}(n^2)$ in the worst case.

Before updating $\boldsymbol{Q^-}$ and $\boldsymbol{Q^+}$, first we need to establish the elements affected by the edge swap. Between T and T', all q_{st}^- and q_{st}^+ for $s \in T_1, t \in \{T_2 \backslash \varphi_2 \cup T_1 \backslash \varphi_1\}$ and $s \in T_2, t \in \{T_1 \backslash \varphi_1 \cup T_2 \backslash \varphi_2\}$ remains unchanged. In other words, we only need to update the elements q_{st}^- and q_{st}^+ for $s \in T_1 \cup T_2, t \in \varphi_1 \cup \varphi_2$. Removal of any node $t \in \varphi_1$ and all of its edges

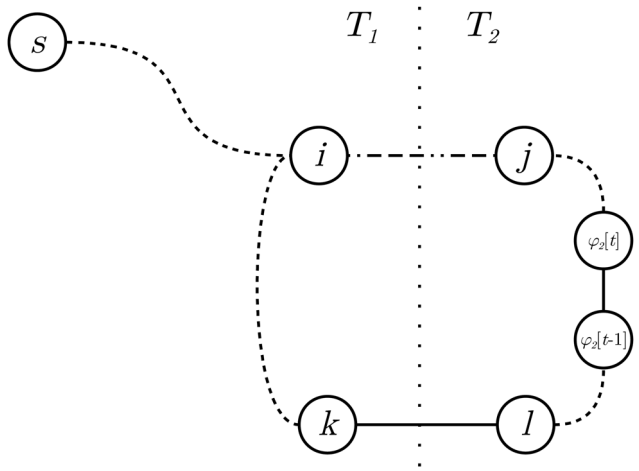

FIG. 4. After the edge swap between e_{ij} and e_{kl}, the predecessors of nodes $t \in T_1 \cup T_2 \backslash \varphi_2$ in the path $P(s, t)$ do not change. However, we update the predecessors such that the predecessor of $l = \varphi_2[1]$ becomes k, the predecessor $\varphi_2[2]$ becomes $\varphi_2[1]$; in general the predecessor of $\varphi_2[t]$ becomes $\varphi_2[t-1]$.

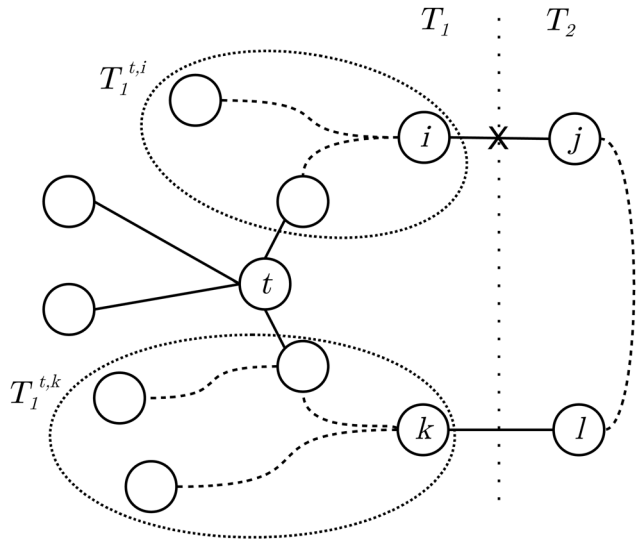

FIG. 5. Subtrees $T_1^{t,i}$ and $T_1^{t,k}$ created by removing $t \in \varphi_1$ and all of its edges from T_1.

Algorithm 2. Predecessor Update

Input: T, T_1, T_2, e_{kl}
$pred(s, l) = k, \quad s \in T_1$
for $t = 2, \ldots, |\varphi_2|$ **do**
  for all $s \in T_1$ **do**
    $pred(s, \varphi_2[t]) = \varphi_2[t-1]$
  end for
end for

from T_1 will create multiple subtrees. Let $T_1^{t,k}$ be the subtree containing k after the removal of $t \in \varphi_1$; similarly let $T_1^{t,i}$ be the subtree containing i. A depiction of these subtrees is given in Figure 5. The other subtrees are of no consequence since the demands from/to those nodes to/from t and its children are not affected by the edge swap.

For any node $s \in T_1, q_{sl}^{-'}$ is equal to q_{sj}^{-} since the nodes l and its children in T' are the same nodes as j and its children in T. Recall that $\varphi_2[1] = l$. Consequently, $q_{s,\varphi_2[2]}^{-'}$ can be found by calculating the difference between the total demand from s to T_2 and $q_{s,\varphi_2[1]}^{-}$, which is given by $q_{sj}^{-} - q_{s,\varphi_2[1]}^{-}$. As we iterate over the cycle φ_2, the general expression becomes, $q_{s,\varphi_2[t]}^{-'} = q_{sj}^{-} - q_{s,\varphi_2[t-1]}^{-}$. The update for $\boldsymbol{Q}^{+}$ is the same as for $\boldsymbol{Q}^{-}$ (and can be computed and updated at the same time as the $\boldsymbol{Q}^{-}$ update).

For $t \in \varphi_1, q_{st}^{-'}$ depends on the domain of s. If $s \in T_1^{t,i}$, then the total demand from s to T_2 must traverse t in T' though this was not the case for T; therefore $q_{st}^{-'}$ becomes q_{st}^{-} plus the total demand from s to T_2 (q_{sj}^{-}). If $s \in T_1^{t,k}$, then the total demand from s to T_2 no longer traverses t in T', so $q_{st}^{-'}$ becomes q_{st}^{-} minus q_{sj}^{-}. Finally, if $s \in T_1/(T_1^{t,k} \cup T_1^{t,i})$, the total demand from s to T_2 has to traverse t for both T and T' so $q_{st}^{-'} = q_{st}^{-}$. The procedure for updating $\boldsymbol{Q}^{-}$ and $\boldsymbol{Q}^{+}$ is summarized in Algorithm 3; as in Algorithm 2, we only provide the update for $s \in T_1$. In the worst case, we do $\mathcal{O}(n^2)$ updates for $\boldsymbol{Q}^{-}$ and $\boldsymbol{Q}^{+}$.

Algorithm 3. $\boldsymbol{Q}^{-}, \boldsymbol{Q}^{+}$ Update

Input: $T, T_1, T_2, e_{ij}, e_{kl}$
for all $s \in T_1$ **do**
  $q_{sl}^{-'} = q_{sj}^{-}$
  $q_{sl}^{+'} = q_{sj}^{+}$
  for $t = 2, \ldots, |\varphi_2|$ **do**
    $q_{s,\varphi_2[t]}^{-'} = q_{sj}^{-} - q_{s,\varphi_2[t-1]}^{-}$
    $q_{s,\varphi_2[t]}^{+'} = q_{sj}^{+} - q_{s,\varphi_2[t-1]}^{+}$
  end for
end for
for all $t \in \varphi_1$ **do**
  for all $s \in T_1^{t,i}$ **do**
    $q_{st}^{-'} = q_{st}^{-} + q_{sj}^{-}$
    $q_{st}^{+'} = q_{st}^{+} + q_{sj}^{+}$
  end for
  for all $s \in T_1^{t,k}$ **do**
    $q_{st}^{-'} = q_{st}^{-} - q_{sj}^{-}$
    $q_{st}^{+'} = q_{st}^{+} - q_{sj}^{+}$
  end for
end for

3. INITIAL SOLUTION AND EDGE-SWAPPING ALGORITHMS

In this section, we discuss our initial solution procedure, and the local search and tabu search algorithms that use the tree–nontree edge swap neighborhood.

3.1. Initial Solution Procedure

Suppose we select a color c. Let ε_c^k be the number of edges of color c connected to node k. First we select the node s such that it has the maximum ε_c^s (breaking ties arbitrarily). Then we start the breadth-first search (BFS) from s only using edges with color c. If it is possible to span the network only using color c we will have an initial solution at the end of the BFS (as well as the optimal solution since it will have zero cost). If it is not possible to span the entire network via BFS using a single color, we add the remaining nodes one by one to the partially constructed tree by selecting the edge that will create the minimum increase in the total reload cost (i.e., in a greedy fashion). At the end of this process, we will have a spanning tree. Let C denote the total number of colors in the graph. We apply this initial solution procedure C times, once for each color. Thus, we obtain up to C different initial solutions. We then select the tree with the least total reload cost and report it as the *Initial Solution* (IS).

3.2. Local Search Algorithm

Given a feasible spanning tree T, our local search algorithm identifies the best edge swap and (if it results in an improvement) updates T iteratively until no improvement can be found in the tree–nontree edge swap neighborhood. Algorithm 4 outlines the local search procedure for a given initial feasible spanning tree T. We use the local search in a multistart framework, applying it once to each of the C initial solutions found by our initial solution procedure. We then select the tree with the least total reload cost and report it as the *Local Search Solution* (LS).

Algorithm 4. Local Search Procedure for a given T

Input: G, T
Output: T
$\Delta = -\infty$
while $\Delta < 0$ **do**
  Identify (e^{in}, e^{out}) the *best tree-nontree edge swap* for T; where e^{in} and e^{out} denote the incoming and outgoing edge respectively.
  $T' = T \cup e^{in} \setminus e^{out}$
  $\Delta = \delta(T, T')$
  if $\Delta < 0$ **then**
    $T = T'$
    Update predecessors and $\boldsymbol{R_p}, \boldsymbol{Q^-}, \boldsymbol{Q^+}$
  end if
end while
return T

3.3. Tabu Search Algorithm

In our tabu search algorithm, the process of exploring the neighborhood and updating the preprocessed information are identical to that of the local search algorithm. However, in the tabu search algorithm, after a tree edge $e^{out} \in T$ is swapped with a nontree edge $e^{in} \in E\backslash T$, e^{out} and e^{in} are declared tabu, that is, e^{out} cannot reenter to the current tree and e^{in} cannot leave the current tree for the next κ edge swaps. Even though using edges as tabu elements has its advantages, it is possible to reject an edge swap that might improve the best known solution. To remedy that, we introduce an aspiration criterion; a tabu swap is accepted only if it improves the best known solution. Therefore, in one iteration of the tabu search algorithm, the best edge swap is implemented if it improves the best known solution, otherwise the best nontabu edge swap is implemented.

Given a feasible spanning tree T, the tabu search algorithm identifies an edge swap as described and updates T iteratively until it makes X consecutive edge swaps without improving the best known solution. Observe that with the aspiration criterion, the tabu search algorithm makes the same tree–nontree edge swaps as the local search algorithm until the first nonimproving iteration. Thus, it is not possible for the tabu search algorithm to terminate with an inferior solution to that of the local search algorithm. As with the local search procedure, we use the tabu search procedure in a multistart framework, applying it once to each of the C initial solutions found by our initial solution procedure. We then select the tree with the least total reload cost and report it as the *Tabu Search Solution* (TS).

4. SINGLE-SOURCE VARIANTS

In some telecommunications, applications demand only occurs between a single node (e.g., a central office) and the other nodes in the network. Further, when equipment costs are high (as in the case of telecommunications applications) and predominate other (variable) costs one is interested in minimizing the fixed installation costs. This motivates two special variants of the RCSTP—the single-source RCSTP and the single-source-fixed RCSTP—that we discuss in this section.

4.1. Single-Source RCSTP

The single-source RCSTP is a special case of the RCSTP introduced in Gamvros et al. [6]. In the single-source RCSTP, a node $s \in V$ is given as the source node and there is demand only from node s to every other node in V, $d_{sj} \geq 0$, for all $j \in V$ and $d_{ij} = 0$, for all $i \neq s, j \in V$. Gamvros et al. [6] show that the single-source RCSTP is also NP-hard. The earlier initial solution procedure remains valid and is used to generate C initial solutions. In addition, we introduce another initial solution procedure to take advantage of one of the problem characteristics. Specifically, there are no reloads between edges connected to node s since there is no demand that traverses through node s (i.e., uses node s as an intermediate node between its origin and destination). Therefore, unlike the RCSTP, it is possible to obtain a zero cost spanning tree without having to span the graph with a single color, so long as each subtree rooted at s is spanned by a single color. We

first start by adding all edges connected to s to the initial tree T. Then we apply a BFS starting from each of these nodes connected to s using the same color as the edge connecting it with s (i.e., if e_{si} has color c_1 we apply a BFS from node i only considering edges with color c_1). If the resulting tree is a spanning tree, the procedure terminates. Otherwise we add the edge that will create the minimum increase in reload cost (breaking ties arbitrarily). We refer to this as a greedy step. We continue by applying a BFS from the node that was just added to the tree using the color of the edge that was just added to the tree. We refer to this as the BFS step. We continue in this fashion alternating between a greedy step and the BFS step until we obtain a spanning tree. The best of these initial $C+1$ solutions is reported as the initial solution (IS). The local search algorithm and the tabu search algorithm for the single-source problem remain unchanged (as the edge swap neighborhood is still valid and the algorithm is designed to work with any set of demands). The only difference is that they are now run in a multistart framework, applying them once to each of the $C+1$ initial solutions.

4.2. *Single-Source-Fixed RCSTP*

In the single-source-fixed RCSTP, a node $s \in V$ is given as the source node and the set of feasible solutions are spanning trees rooted at node s. For a given spanning tree T rooted at s and a node $i \in V\backslash\{s\}$, the total reload cost incurred in node i is calculated as follows. Consider the color $c_{pred(s,i),i}$ of the incoming edge into node i (on the path from the node s to node i). Consider the set of colors Γ_i of all the other outgoing edges incident to node i. For each color $c \in \Gamma_i$ with $c \neq c_{pred(s,i),i}$, a fixed reload cost $r^i(c_{pred(s,i),i}, c)$ is occurred. Then the total fixed reload cost $R_f(T,s)$ for spanning tree T rooted at s is given by

$$R_f(T,s) = \sum_{\substack{i \in T \\ i \neq s}} \sum_{c \in \Gamma_i \atop c \neq c_{pred(s,i),i}} r^i(c_{pred(s,i),i}, c).$$

Observe that unlike the RCSTP reload costs do not depend on demands (in fact there are no demands in the problem). To emphasize this difference in how the reload costs are calculated, we use the term *fixed* in the name of the problem. Further, notice that at each node a reload cost is only incurred once for each color change that takes place at the node. The objective in the single-source-fixed RCSTP is to minimize the total fixed reload cost incurred.

The procedures proposed for the single-source RCSTP also apply to the single-source-fixed RCSTP save for the cost calculation step. Consequently, in the initial solution procedure, we generate $C+1$ initial solutions and report the best one as the initial solution. For the edge-swap neighborhood, the calculations are slightly different because of the fixed costs. We elaborate on the calculation of the change in reload cost in a tree–nontree edge swap. Consider a spanning tree T and the edge swap $e_{ij} \in T$ and $e_{kl} \in E/T$ that results in the spanning tree T' after the swap. We would like to calculate $\delta_f(T,T',s) = R_f(T',s) - R_f(T,s)$. The removal of e_{ij} from

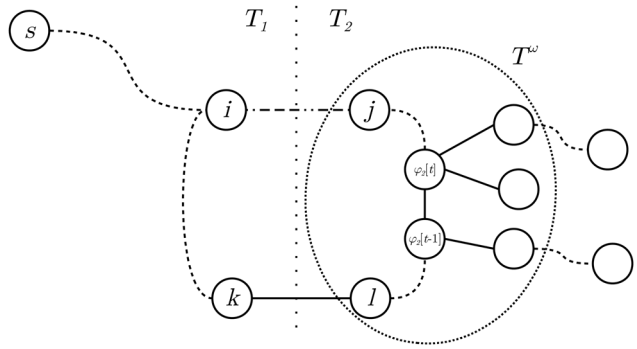

FIG. 6. Subtree T^ω created by the nodes in φ_2 and their immediate successors.

T creates two subtrees T_1 and T_2 as discussed previously. Without loss of generality, assume $i,k,s \in T_1$ and $j,l \in T_2$. Let $\omega = \{t | pred(s,t) \in \varphi_2\} \cup \varphi_2$ where φ_2 is the list of nodes in the cycle in T_2 created by adding e_{kl} to T. In other words, ω is the set of nodes in φ_2 and their immediate successors. Then, we define T^ω as the subtree of T formed by the nodes in ω. Figure 6 illustrates T^ω. Let $T^{\omega,i} = T^\omega \cup e_{ij}$ and $T^{\omega,k} = T^\omega \cup e_{kl}$. Further, let $\Gamma_i(T,s)$ represent the set of colors incident to node i in the tree T with source s, when considering all edges incident to node i except for $c_{pred(s,i),i}$. Then, we have

$$\begin{aligned}\delta_f(T,T',s) = &(R_f(T^{\omega,k},k) + R^k(T_1,e_{kl})) \\ &- (R_f(T^{\omega,i},i) + R^i(T_1,e_{ij}))\end{aligned}$$

where

$$R^k(T_1,e_{kl}) = \begin{cases} 0 & \text{if } \exists c \in \Gamma_k(T_1,s) \text{s.t.} \\ & \quad c = c_{kl}, \\ r^k(c_{pred(s,k),k}, c_{kl}) & \text{otherwise.} \end{cases}$$

After the edge swap, the only affected part of the tree is $T^\omega \cup \{e_{ij}, e_{kl}\}$. We capture the difference in total fixed reload costs between T and T' by calculating only the costs related to the affected subtree. For any T and s, $R_f(T,s)$ can easily be calculated by applying a BFS starting from node s. We store and update predecessor information as in section 2. One local search iteration for the single-source-fixed RCSTP is $\mathcal{O}(mn^2)$ as in the RCSTP since evaluating one edge swap is $\mathcal{O}(n)$. As before, the local search algorithm and the tabu search algorithm are run in a multistart framework, applying them once to each of the $C+1$ initial solutions.

5. COMPUTATIONAL RESULTS

We conducted a large set of computational experiments to examine the quality of the heuristics. They contain instances used in Gamvros et al. [6] and Khalil and Singh [9], as well as additional test instances that we generated. As parameters of the tabu search algorithm, we used $\kappa = n/4$ and $X = 1000$, that is, once declared tabu, an edge cannot reenter or leave the tree for $n/4$ edge swaps unless it satisfies the aspiration criterion, and the algorithm terminates after 1000 consecutive edge swaps without improving the best solution. The

heuristics are coded in C++ and all computations are conducted on a computer with an Intel Core i7-2600 CPU @ 3.40GHz and 16 GB RAM running Windows 7.

The Gamvros et al. [6] instances consist of four types. The types differ based on whether they have unit reload costs (reload costs between all pairs of colors are set to 1) or nonunit reload costs, and whether they have a single-source (in single-source problems all demands that do not originate at the designated source node s are set to 0) or demands between all pairs. Thus, there are four different types of instances that we refer to as (i) unit reload cost, (ii) nonunit reload cost, (iii) single-source unit reload cost, and (iv) single-source nonunit reload cost (if we do not use the qualifier single source then all pairs of demands are permitted). For all four types, the demand between pairs of nodes is set to 1 (i.e., this set of instances do not contain nonunit demand). For single-source instances, the demand from s to all other nodes is 1. We should note that [6] actually contains 55 unit reload cost, 20 nonunit reload cost, 75 single-source unit reload cost, and 40 single-source nonunit reload cost instances. After a preliminary computational study on these instances, we observed that for the instances that have a large number of nodes and edges but few colors (e.g., 50 nodes, 300 edges, and 3 colors) it is more than likely that the whole network can be spanned by a single color thus providing a zero total reload cost spanning tree. These instances tend to be easy for our initial heuristic to solve (though they seem to be hard for CPLEX), and consequently we excluded these somewhat trivial instances from our computational experiments. Instead we generated additional instances (as described in [6]) that have a larger number of colors as the number of nodes and edges in the instance increase. By using a subset of the Gamvros et al. [6] instances and the additional instances, we generated and obtained a set of 105 test instances for each of the four types of problems. These 105 instances are labeled as NxEyCz where x, y, and z represent the number of nodes, number of edges, and number of colors, respectively. Note that the topology of the instance remains the same across the four types of problems (i.e., the graphs for the instances are identical); we simply change the reload costs and/or demands across the four problem types. In our tables, we identify in bold letters the subset of instances that are reported upon in [6].

In our computational experiments, we compare the tabu search solution to the local search solution and initial solution. This provides one measure of the benefit of the tabu search procedure. We also compare the heuristic solutions to the optimal solution when available.

5.1. *Gamvros et al. Instances*

Tables 1–4 report on the four types of Gamvros et al. [6] instances. The first column in these tables identifies the instance set. The second column indicates the number of instances contained in the set. Our data set has five instances for each set. For each set of instances, we report on the average value of the objective function, and the number of optimal solutions obtained for each of the heuristics. Specifically, IS represents the solution found by the initial solution procedure, LS represents the solution found by the local search procedure, and TS represents the solution found by the tabu search procedure. Recall, our initial solution procedure is run multiple times (it is applied $C + 1$ times for the single source problems and C times for the all-pairs problems) and the local search and tabu search algorithm are run in a multistart framework. To ascertain the optimal solution to the instances, we note that if TS (or for that matter IS or LS) has cost zero, then it must be an optimal solution. For the problems where tabu search does not provide a zero cost objective (ZCO), we apply the colored graph formulation from Gamvros et al. [6] and solve it using CPLEX (version 12.5) with a 6-h time limit (our CPLEX implementation of the colored graph formulation is in C++). Taken together, instances that have ZCO and instances that CPLEX is able to solve to optimality provide us a set of instances for which we know the optimal solution. Interestingly, for instances that CPLEX did not optimally solve in 6h, it did not provide useful upper bounds (TS was always significantly better) or lower bounds (they were generally equal to the trivial lower bound of 0). The set of columns associated with the "Average Objective Value" provide the average across the five instances for IS, LS, and TS; the average under the column OPT is only provided when we know the optimal solution for all five instances. The set of columns associated with "Number Optimal" provide the number of optimal solutions found by IS, LS, and TS, the number of instances with ZCO, and the number of problems that CPLEX solves to optimality across the five instances.

To evaluate the benefit of tabu search, we also track the improvement in the solution from IS to LS, and from LS to TS. These are reported in the columns associated with "Improvement." We provide both the number of instances (#) for which the solution is improved (from IS to LS and from LS to TS) as well as the percentage (%) improvement (the average across five instances in each row). Finally, we provide the average running time for the tabu search algorithm as well as for CPLEX. The CPLEX average running time is only across instances that CPLEX solved to optimality.

Table 1 presents results for the unit reload cost instances. Out of 105 instances, we know the optimal solutions for 48 instances (6 are ZCO and 42 optimal solutions are obtained through CPLEX). Because the colored graph formulation from Gamvros et al. [6] is a flow-based formulation, it becomes intractable very quickly (in terms of number of variables and constraints) for all-pairs problems and CPLEX is unable to solve instances with 50 or more nodes, while TS rapidly finds solutions for these instances. For the 48 instances for which we know the optimal solution, IS is optimal for 19 instances, LS is optimal for 34 instances, while TS is optimal in all 48 instances. Taking a look at improvements, there is an IS to LS improvement in 70 out of 105 instances with an average improvement of 5.66%, while there is an LS to TS improvement in 61 out of 105 instances with an average improvement of 1.72%. As the instances get harder, i.e., the number of nodes and colors increase, tabu search improves upon local search in a greater number of instances.

TABLE 1. Unit reload cost instances.

Instance Name	Number of instances	Average objective value				Number optimal					Improvement				Average Run time (s)	
											IS to LS		LS to TS			
		IS	LS	TS	OPT	IS	LS	TS	ZCO	CPLEX	#	%	#	%	TS	CPLEX
N10E25C3	5	37.6	12.0	12.0	12.0	2	5	5	2	3	3	42.50%	0	0.00%	0.05	0.75
N10E25C5	5	35.6	22.8	22.4	22.4	2	4	5	1	4	2	21.82%	1	0.95%	0.10	0.52
N10E25C7	5	56.0	52.4	51.6	51.6	2	4	5	0	5	3	5.60%	1	1.54%	0.16	0.66
N15E50C5	5	76.4	70.4	67.6	67.6	2	3	5	0	5	3	5.14%	2	2.54%	0.35	13.52
N15E50C7	5	124.4	118.0	115.6	115.6	1	2	5	0	5	4	5.25%	3	2.18%	0.47	16.19
N15E50C9	5	181.6	165.2	159.2	159.2	0	2	5	0	5	5	9.19%	3	3.39%	0.57	25.82
N20E100C5	5	36.0	36.0	36.0	36.0	5	5	5	0	5	0	0.00%	0	0.00%	1.03	11775.66
N20E100C7	5	83.2	82.0	80.0	80.0	2	3	5	0	5	1	1.11%	2	1.68%	1.43	1837.56
N20E100C9	5	162.4	149.6	147.2	147.2	0	3	5	0	5	5	7.23%	2	1.26%	1.86	3241.34
N50E300C5	5	248.4	247.2	243.2	–	0	0	0	0	0	1	0.42%	3	1.22%	9.83	–
N50E300C7	5	639.6	620.4	618.0	–	0	0	0	0	0	3	2.39%	2	0.34%	15.89	–
N50E300C9	5	1541.6	1481.6	1414.4	–	0	0	0	0	0	5	3.85%	5	4.38%	25.04	–
N50E300C11	5	1944.8	1858.0	1752.0	–	0	0	0	0	0	5	4.35%	5	5.26%	30.16	–
N75E695C7	5	174.4	174.4	172.8	–	2	2	2	2	0	0	0.00%	2	0.46%	39.52	–
N75E695C9	5	927.2	920.4	909.6	–	0	0	0	0	0	4	0.84%	4	1.08%	92.12	–
N75E695C11	5	2061.2	2028.0	1970.0	–	0	0	0	0	0	4	1.48%	4	2.80%	127.89	–
N75E695C13	5	2994.0	2870.4	2829.2	–	0	0	0	0	0	5	4.10%	5	1.38%	179.98	–
N100E1225C9	5	738.4	736.4	732.8	–	1	1	1	1	0	2	0.18%	3	0.44%	199.80	–
N100E1225C11	5	1668.8	1661.2	1638.8	–	0	0	0	0	0	5	0.43%	4	1.12%	306.03	–
N100E1225C13	5	2488.0	2453.6	2404.8	–	0	0	0	0	0	5	1.32%	5	2.05%	469.71	–
N100E1225C15	5	3489.2	3430.8	3360.4	–	0	0	0	0	0	5	1.67%	5	1.98%	656.19	–

TABLE 2. Single-source unit reload cost instances.

Instance Name	Number of instances	Average objective value				Number optimal					Improvement				Average Run time (s)	
											IS to LS		LS to TS			
		IS	LS	TS	OPT	IS	LS	TS	ZCO	CPLEX	#	%	#	%	TS	CPLEX
N10E25C3	5	1.0	0.6	0.4	0.4	3	4	5	3	2	2	30.00%	1	20.00%	0.04	0.03
N10E25C5	5	1.2	0.6	0.6	0.6	4	5	5	4	1	1	20.00%	0	0.00%	0.03	0.18
N10E25C7	5	2.4	2.2	2.0	2.0	3	4	5	1	4	1	20.00%	1	6.67%	0.15	0.02
N15E50C5	5	2.0	1.6	1.4	1.4	3	4	5	0	5	1	13.33%	1	10.00%	0.44	0.19
N15E50C7	5	3.4	2.6	2.6	2.6	3	5	5	1	4	2	26.67%	0	0.00%	0.42	0.06
N15E50C9	5	4.6	4.0	4.0	4.0	2	5	5	0	5	3	15.83%	0	0.00%	0.61	0.09
N20E100C5	5	1.2	0.0	0.0	0.0	1	5	5	5	0	4	80.00%	0	0.00%	0.00	–
N20E100C7	5	2.4	0.6	0.2	0.2	0	3	5	4	1	5	82.67%	2	30.00%	0.32	0.17
N20E100C9	5	3.0	1.6	0.8	0.8	1	2	5	2	3	3	38.67%	3	50.00%	1.18	0.20
N50E300C5	5	2.0	1.8	0.0	0.0	0	0	5	5	0	1	6.67%	5	100.00%	0.17	–
N50E300C7	5	5.4	3.4	1.0	1.0	0	1	5	2	3	5	45.62%	4	58.67%	11.85	50.55
N50E300C9	5	13.4	11.0	6.8	6.8	0	0	5	0	5	4	19.00%	5	43.77%	22.43	12.74
N50E300C11	5	10.4	9.4	7.2	7.2	1	1	5	0	5	3	9.34%	4	25.32%	22.51	5.63
N75E695C7	5	3.0	0.6	0.0	0.0	1	2	5	5	0	4	62.00%	3	60.00%	0.83	–
N75E695C9	5	6.0	4.2	0.0	0.0	0	0	5	5	0	4	33.24%	5	100.00%	18.83	–
N75E695C11	5	7.6	6.6	0.0	0.0	0	0	5	5	0	2	11.11%	5	100.00%	3.73	–
N75E695C13	5	14.4	11.2	2.8	-	0	0	3	2	1	5	21.55%	5	78.48%	96.80	207.37
N100E1225C9	5	3.8	2.2	0.0	0.0	0	1	5	5	0	5	48.33%	4	80.00%	1.57	–
N100E1225C11	5	5.8	4.0	0.0	0.0	0	0	5	5	0	5	32.86%	5	100.00%	4.39	–
N100E1225C13	5	10.2	7.6	0.0	0.0	0	0	5	5	0	3	20.00%	5	100.00%	9.67	–
N100E1225C15	5	15.4	12.6	0.8	-	0	0	2	2	1	5	16.05%	5	93.81%	486.72	5280.61

Table 2 presents results for the single-source unit reload cost instances. This problem turns out to be somewhat easier than the all-pairs problem. Out of 105 instances, we know the optimal solutions for 101 instances (61 are ZCO and 40 optimal solutions are obtained via CPLEX). In only four instances (twice for N75E695C13 and twice for N100E1225C15) was CPLEX unable to find the optimal solution in the 6-h time limit. For the 101 instances for which we know the optimal solution, IS is optimal in 22 instances, LS is optimal in 42 instances, and TS is optimal in 100 instances. Turning our attention to improvements, the benefits of local search and tabu search become even more apparent (especially

TABLE 3. Nonunit reload cost instances.

Instance Name	Number of instances	Average objective value				Number optimal					Improvement				Average Run time (s)	
											IS to LS		LS to TS			
		IS	LS	TS	OPT	IS	LS	TS	ZCO	CPLEX	#	%	#	%	TS	CPLEX
N10E25C3	5	62.4	36.8	36.8	36.8	3	5	5	2	3	2	22.50%	0	0.00%	0.06	0.75
N10E25C5	5	134.0	95.2	95.2	90.0	1	4	4	1	4	3	26.22%	0	0.00%	0.10	0.53
N10E25C7	5	108.0	90.4	90.4	90.4	1	5	5	0	5	4	13.88%	0	0.00%	0.17	0.48
N15E50C5	5	213.2	170.8	170.8	166.0	3	4	4	0	5	2	13.68%	0	0.00%	0.36	5.05
N15E50C7	5	447.6	337.6	334.4	334.4	0	4	5	0	5	5	23.06%	1	1.42%	0.50	6.09
N15E50C9	5	586.8	411.6	411.6	402.4	0	2	2	0	5	5	29.18%	0	0.00%	0.67	16.21
N20E100C5	5	35.2	20.8	20.8	20.8	4	5	5	3	2	1	20.00%	0	0.00%	0.53	1474.85
N20E100C7	5	178.4	176.8	176.0	176.0	3	4	5	0	5	2	0.66%	1	0.28%	1.39	1067.37
N20E100C9	5	372.8	361.6	355.2	350.4	1	3	4	0	5	4	2.71%	1	1.72%	1.87	653.54
N50E300C5	5	459.2	459.2	459.2	–	0	0	0	0	0	0	0.00%	0	0.00%	9.38	–
N50E300C7	5	1266.8	1184.0	1175.2	–	0	0	0	0	0	4	4.33%	1	0.51%	13.92	–
N50E300C9	5	3270.8	2465.6	2441.2	–	0	0	0	0	0	5	20.40%	2	1.15%	18.71	–
N50E300C11	5	3740.8	2847.2	2816.0	–	0	0	0	0	0	5	22.65%	3	0.93%	23.68	–
N75E695C7	5	378.0	378.0	376.4	–	2	2	2	2	0	0	0.00%	2	0.18%	39.11	–
N75E695C9	5	1310.8	1276.8	1270.8	–	0	0	0	0	0	4	1.88%	2	0.47%	78.66	–
N75E695C11	5	2332.4	2261.6	2252.8	–	0	0	0	0	0	4	3.40%	3	0.28%	95.73	–
N75E695C13	5	3838.8	3337.2	3294.0	–	0	0	0	0	0	5	12.34%	4	1.22%	116.61	–
N100E1225C9	5	1056.8	1052.8	1049.6	–	1	1	1	1	0	3	0.30%	3	0.28%	187.03	–
N100E1225C11	5	2107.2	2104.4	2100.4	–	0	0	0	0	0	1	0.11%	4	0.25%	276.01	–
N100E1225C13	5	2698.4	2691.6	2684.4	–	0	0	0	0	0	3	0.22%	4	0.26%	344.55	–
N100E1225C15	5	3658.4	3625.2	3612.4	–	0	0	0	0	0	5	0.87%	4	0.33%	437.02	–

TABLE 4. Single-source nonunit reload cost instances.

Instance Name	Number of instances	Average objective value				Number optimal					Improvement				Average Run time (s)	
											IS to LS		LS to TS			
		IS	LS	TS	OPT	IS	LS	TS	ZCO	CPLEX	#	%	#	%	TS	CPLEX
N10E25C3	5	4.6	2.0	1.0	1.0	2	4	5	3	2	3	40.00%	1	20.00%	0.04	0.30
N10E25C5	5	3.2	2.0	2.0	2.0	4	5	5	4	1	1	20.00%	0	0.00%	0.03	0.33
N10E25C7	5	7.4	3.8	3.8	3.6	1	4	4	1	4	4	48.41%	0	0.00%	0.15	0.16
N15E50C5	5	5.2	4.2	3.6	3.6	3	3	5	0	5	2	18.00%	2	20.00%	0.43	0.37
N15E50C7	5	13.2	5.4	5.4	5.4	0	5	5	1	4	5	51.72%	0	0.00%	0.45	0.30
N15E50C9	5	12.2	7.8	7.0	7.0	1	3	5	0	5	3	32.83%	2	7.45%	0.64	0.35
N20E100C5	5	1.0	0.2	0.0	0.0	2	4	5	5	0	3	50.00%	1	20.00%	0.00	–
N20E100C7	5	2.8	1.6	1.2	1.2	1	4	5	3	2	4	56.67%	1	20.00%	0.66	0.32
N20E100C9	5	7.8	7.0	6.0	6.0	1	1	5	0	5	3	16.67%	4	22.64%	1.97	0.20
N50E300C5	5	5.6	3.2	0.0	0.0	0	0	5	5	0	2	22.22%	5	100.00%	0.43	–
N50E300C7	5	11.6	7.0	1.4	1.4	0	1	5	2	3	4	43.97%	4	64.68%	13.29	35.50
N50E300C9	5	30.4	18.6	10.0	9.8	0	0	4	0	5	5	35.84%	5	50.62%	21.52	22.85
N50E300C11	5	21.0	15.8	9.2	9.2	0	0	5	0	5	5	20.96%	5	42.07%	24.23	6.17
N75E695C7	5	4.0	1.6	0.0	0.0	1	2	5	5	0	4	51.67%	3	60.00%	0.43	–
N75E695C9	5	11.4	7.6	0.0	0.0	0	0	5	5	0	4	30.50%	5	100.00%	10.00	–
N75E695C11	5	10.6	9.0	0.0	0.0	0	0	5	5	0	3	17.71%	5	100.00%	3.70	–
N75E695C13	5	20.2	11.8	2.6	2.6	0	0	5	2	3	5	42.02%	5	81.72%	94.90	883.05
N100E1225C9	5	7.2	3.6	0.0	0.0	0	1	5	5	0	5	51.71%	4	80.00%	1.78	–
N100E1225C11	5	8.4	6.4	0.0	0.0	0	0	5	5	0	4	21.56%	5	100.00%	2.90	–
N100E1225C13	5	12.4	8.0	0.0	0.0	0	0	5	5	0	5	34.97%	5	100.00%	18.53	–
N100E1225C15	5	19.8	13.4	0.8	-	0	0	3	3	0	5	28.63%	5	94.33%	403.69	–

taking into account the percentage improvements). There is an IS to LS improvement in 68 out of 105 instances with an average improvement of 31.09%, while there is an LS to TS improvement in 63 out of 105 instances with an average improvement of 50.32%. As in the all-pairs instances, the harder the instances get the greater the number of instances tabu search improves upon local search.

Table 3 presents results for the nonunit reload cost instances. Out of 105 instances, we know the optimal solutions for 48 instances (9 are ZCO and 39 optimal solutions are obtained via CPLEX). As in the unit reload cost instances, once the number of nodes reaches 50, CPLEX fails to find the optimal solution in 6h. For the 48 instances for which we know the optimal solution, IS is optimal in 19 instances, LS

TABLE 5. Single-source-fixed cost unit reload cost instances.

Instance Name	Number of instances	Average objective value				Number optimal					Improvement				Average Run time (s)	
											IS to LS		LS to TS			
		IS	LS	TS	OPT	IS	LS	TS	ZCO	CPLEX	#	%	#	%	TS	CPLEX
N10E25C3	5	0.8	0.6	0.4	0.4	3	4	5	3	2	1	20.00%	1	20.00%	0.07	0.04
N10E25C5	5	0.6	0.4	0.4	0.4	4	5	5	4	1	1	20.00%	0	0.00%	0.06	0.05
N10E25C7	5	1.6	1.4	1.2	1.2	3	4	5	1	4	1	20.00%	1	10.00%	0.25	0.04
N15E50C5	5	1.4	1.2	1.2	1.2	4	5	5	0	5	1	10.00%	0	0.00%	0.69	0.22
N15E50C7	5	2.6	1.4	1.0	1.0	1	3	5	1	4	3	36.67%	2	20.00%	0.80	0.22
N15E50C9	5	2.6	2.2	2.0	2.0	3	4	5	0	5	2	15.00%	1	6.67%	1.07	0.37
N20E100C5	5	0.8	0.0	0.0	0.0	1	5	5	5	0	4	80.00%	0	0.00%	0.01	–
N20E100C7	5	1.4	0.6	0.2	0.2	1	3	5	4	1	3	53.33%	2	40.00%	0.52	3.82
N20E100C9	5	2.2	1.4	0.6	0.6	1	2	5	2	3	3	30.00%	3	50.00%	2.23	2.50
N50E300C5	5	1.8	1.6	0.0	0.0	0	0	5	5	0	1	6.67%	5	100.00%	1.42	–
N50E300C7	5	4.4	2.2	0.6	0.6	0	1	5	2	3	5	55.00%	4	56.67%	79.12	6990.34
N50E300C9	5	5.6	4.8	2.2	1.8	0	0	3	0	5	4	13.05%	5	54.33%	124.86	7528.24
N50E300C11	5	6.8	6.2	2.6	2.4	0	0	4	0	5	2	7.50%	5	57.52%	35.16	585.40
N75E695C7	5	2.2	0.8	0.0	0.0	1	2	5	5	0	4	53.33%	3	60.00%	1.45	–
N75E695C9	5	4.4	3.0	0.0	0.0	0	0	5	5	0	4	31.00%	5	100.00%	33.63	–
N75E695C11	5	4.4	4.0	0.0	0.0	0	0	5	5	0	2	9.00%	5	100.00%	36.98	–
N75E695C13	5	7.6	6.8	1.2	-	0	0	2	2	0	3	9.44%	5	83.33%	140.34	–
N100E1225C9	5	2.8	1.4	0.0	0.0	0	1	5	5	0	4	50.00%	4	80.00%	7.10	–
N100E1225C11	5	4.4	3.8	0.0	0.0	0	0	5	5	0	2	10.71%	5	100.00%	21.39	–
N100E1225C13	5	6.0	5.4	0.0	0.0	0	0	5	5	0	3	8.89%	5	100.00%	155.45	–
N100E1225C15	5	8.4	8.0	1.2	-	0	0	0	0	0	2	5.08%	5	85.17%	674.26	–

is optimal in 39 instances, and TS is optimal in 42 instances. Considering improvements, there is an IS to LS improvement in 67 out of 105 instances with an average improvement of 10.4%, while there is an LS to TS improvement in 35 out of 105 instances with an average improvement of 0.44%. While the magnitude of improvement is smaller for nonunit reload cost instances, the number of instances that tabu search improves upon local search increases as the instances get harder.

Table 4 presents results for the single-source nonunit reload cost problem instances. Out of 105 instances, we know the optimal solutions for 103 instances (59 are ZCO and 44 optimal solutions are obtained via CPLEX). In only two instances (both for N100E1225C15), CPLEX could not find the optimal solution in 6h. For the 103 instances for which we know the optimal solution, IS is optimal in 16 instances, LS is optimal in 37 instances, and TS is optimal in 101 instances. Focusing on improvements, there is an IS to LS improvement in 79 out of 105 instances with an average improvement of 35.05%, while there is an LS to TS improvement in 67 out of 105 instances with an average improvement of 51.6%. Once again we observe that as the instances get harder, the number of instances that tabu search improves upon local search increases.

5.2. *Single-Source-Fixed RCSTP Instances*

Gamvros et al. [6] do not conduct any computational experiments on the single-source-fixed RCSTP. Consequently, we use the same 105 single-source unit reload cost instances and 105 single-source nonunit reload cost instances from the RCSTP, and consider the reload costs given as being fixed instead of variable. We also implemented the formulation described in section 6.2 of Gamvros et al. [6] using C++ and CPLEX (version 12.5). Tables 5 and 6 summarize the results for the single-source-fixed RCSTP instances for the unit and nonunit reload cost cases, respectively. They are organized similarly to Table 1.

For the unit reload case discussed in Table 5, out of 105 instances we know the optimal solutions for 97 instances (59 are ZCO and 38 optimal solutions are obtained via CPLEX). In eight instances (three times for N75E695C13 and five times for N100E1225C15), CPLEX could not find the optimal solution in 6h. For the 97 instances for which we know the optimal solution, IS is optimal in 22 instances, LS is optimal in 39 instances, and TS is optimal in 94 instances. Considering improvements, there is an IS to LS improvement in 55 out of 105 instances with an average improvement of 25.94%, while there is an LS to TS improvement in 66 out of 105 instances with an average improvement of 53.51%.

For the nonunit reload cost case discussed in Table 6, out of 105 instances we know the optimal solutions for 98 instances (57 are ZCO and 41 optimal solutions are obtained via CPLEX). In seven instances (three times for N75E695C13 and four times for N100E1225C15), CPLEX could not find the optimal solution in 6h. For the 98 instances for which we know the optimal solution, IS is optimal in 15 instances, LS is optimal in 33 instances, and TS is optimal in 96 instances. Regarding improvements, there is an IS to LS improvement in 77 out of 105 instances with an average improvement of 33.56%, while there is an LS to TS improvement in 72 out of 105 instances with an average improvement of 57.14%.

TABLE 6. Single-source-fixed cost nonunit reload cost instances.

Instance Name	Number of instances	Average objective value				Number optimal					Improvement IS to LS		Improvement LS to TS		Average Run time (s)	
		IS	LS	TS	OPT	IS	LS	TS	ZCO	CPLEX	#	%	#	%	TS	CPLEX
N10E25C3	5	3.0	2.0	1.0	1.0	3	4	5	3	2	1	20.00%	1	20.00%	0.07	0.20
N10E25C5	5	1.6	0.8	0.8	0.8	4	5	5	4	1	1	20.00%	0	0.00%	0.07	0.94
N10E25C7	5	3.6	2.0	2.0	2.0	1	5	5	1	4	4	43.00%	0	0.00%	0.26	0.31
N15E50C5	5	4.2	3.0	2.6	2.6	3	4	5	0	5	2	25.00%	1	13.33%	0.68	1.13
N15E50C7	5	8.2	3.2	2.4	2.4	0	3	5	1	4	5	55.57%	2	28.57%	0.81	1.15
N15E50C9	5	6.6	4.4	3.8	3.8	1	3	5	0	5	3	31.43%	2	12.38%	1.05	1.17
N20E100C5	5	1.0	0.2	0.0	0.0	2	4	5	5	0	3	50.00%	1	20.00%	0.01	–
N20E100C7	5	2.6	1.6	0.6	0.6	0	1	5	3	2	4	34.67%	4	66.67%	1.39	4.38
N20E100C9	5	5.2	3.8	2.4	2.4	0	0	5	0	5	3	22.22%	5	39.00%	3.98	3.60
N50E300C5	5	5.6	3.2	0.0	0.0	0	0	5	5	0	2	22.22%	5	100.00%	0.78	–
N50E300C7	5	9.6	3.6	0.8	0.8	0	1	5	2	3	5	60.67%	4	60.33%	79.53	3883.56
N50E300C9	5	14.0	10.4	3.6	3.2	0	0	3	0	5	4	20.08%	5	67.28%	126.41	6572.87
N50E300C11	5	13.2	10.2	4.0	4.0	0	0	5	0	5	4	21.31%	5	61.21%	33.26	2937.42
N75E695C7	5	3.6	1.2	0.0	0.0	1	2	5	5	0	4	53.33%	3	60.00%	2.52	–
N75E695C9	5	9.4	5.4	0.0	0.0	0	0	5	5	0	5	38.11%	5	100.00%	18.03	–
N75E695C11	5	7.2	5.8	0.0	0.0	0	0	5	5	0	4	16.78%	5	100.00%	11.10	–
N75E695C13	5	11.6	8.4	1.6	-	0	0	2	2	0	5	25.20%	5	83.73%	127.82	–
N100E1225C9	5	5.2	2.8	0.0	0.0	0	1	5	5	0	5	51.00%	4	80.00%	5.96	–
N100E1225C11	5	7.4	4.0	0.0	0.0	0	0	5	5	0	5	46.21%	5	100.00%	11.78	–
N100E1225C13	5	8.0	6.2	0.0	0.0	0	0	5	5	0	4	21.44%	5	100.00%	82.63	–
N100E1225C15	5	11.6	8.2	1.0	-	0	0	1	1	0	4	26.61%	5	87.47%	580.69	–

5.3. *Khalil and Singh Instances*

Khalil and Singh [9] describe two types of benchmark instances: (i) unit demand (where demands between all pairs of nodes are set to 1) and (ii) nonunit demand. The instances are labeled as n*x*cl*y*co*z* where *x*, *y*, and *z* denote the number of nodes, number of clusters in the graph (edges within a cluster have the same color), and the number of colors, respectively. Khalil and Singh [9] conduct their computations on a computer running OpenSUSE 11.1 with an Intel Core 2 Duo processor @ 3GHz and 2GB RAM. None of their instances have ZCO (this is by design of the instances) and none of their instances could be solved with our CPLEX implementation of the colored graph formulation. Hence, for these instances, we do not know the optimal solutions. Since it is the only other paper to date with a heuristic for the RCSTP, we instead focus on comparing our tabu search solution (TS) with their best solution, denoted by KS.

Tables 7 and 8 present these results for the unit and nonunit demand instances, respectively. The first column of these tables identifies the instance name. The next set of columns provides the objective value for KS and TS. The "Improvement" column provides the percentage improvement from KS to TS. A positive value indicates the that TS found a solution with smaller cost than KS, a zero value indicates that the solutions are identical, and a negative value indicates that KS found a better solution than TS. Finally, the "Run Time" columns provide the running times for KS and TS. The running time for KS is only provided for informational purposes and should not directly be compared with that of TS as they are run on *different* computers. For the unit demand instances reported in Table 7, TS improves upon KS in 15 out of 22 instances with an average improvement of 3.48%. These findings are also consistent with the nonunit demand case. As evident from Table 8, TS improves upon KS in 17 out of 22 instances with an average improvement of 3.22%.

TABLE 7. Unit demand instances from Khalil and Singh [9].

Instance Name	Objective value		Improvement KS to TS (%)	Run time (s)	
	KS	TS		KS	TS
n50cl5co3	2514	2514	0.00%	281.04	2.16
n50cl10co3	3184	3184	0.00%	286.46	1.29
n50cl10co6	6490	6490	0.00%	266.04	2.86
n50cl10co9	7518	7326	2.55%	276.71	3.76
n100cl10co3	13564	13564	0.00%	2402.99	12.06
n100cl20co3	19746	18928	4.14%	1608.98	7.49
n100cl10co6	28898	28834	0.22%	1628.46	16.71
n100cl20co6	41220	40500	1.75%	1681.6	11.11
n100cl10co9	43694	43694	0.00%	1634.43	27.50
n100cl20co9	63596	63146	0.71%	1710.89	16.19
n200cl10co3	87654	87654	0.00%	11341.19	63.68
n200cl20co3	50574	45676	9.68%	11569.24	69.62
n200cl10co6	152024	151488	0.35%	10724.31	118.62
n200cl20co6	188228	161666	14.11%	11827.1	112.98
n200cl10co9	194318	193300	0.52%	12014.1	249.66
n200cl20co9	275976	243012	11.94%	11619.65	162.97
n300cl15co3	142188	141858	0.23%	38620.14	331.22
n300cl20co3	150444	130788	13.07%	43877.45	251.75
n300cl15co6	293696	293320	0.13%	43006.62	380.34
n300cl20co6	370178	342664	7.43%	49061.47	294.17
n300cl15co9	484178	484178	0.00%	35516.55	453.48
n300cl20co9	484696	437950	9.64%	54150.14	1082.42

TABLE 8. Nonunit demand instances from Khalil and Singh [9].

Instance Name	Objective value		Improvement KS to TS (%)	Run time (s)	
	KS	TS		KS	TS
n50cl5co3	24121	24121	0.00%	270.56	2.22
n50cl10co3	30226	30226	0.00%	277.88	1.29
n50cl10co6	60274	59835	0.73%	269.57	2.78
n50cl10co9	69952	69636	0.45%	279.7	3.70
n100cl10co3	128844	128844	0.00%	1780.05	11.98
n100cl20co3	183873	178278	3.04%	1707.37	6.24
n100cl10co6	273597	273597	0.00%	1734.1	17.00
n100cl20co6	381746	381421	0.09%	1783.47	11.02
n100cl10co9	414135	415135	−0.24%	1723.02	28.06
n100cl20co9	602329	596920	0.90%	1816.68	17.78
n200cl10co3	838978	838971	0.00%	11149.72	63.68
n200cl20co3	483618	434905	10.07%	11321.18	60.86
n200cl10co6	1449077	1443120	0.41%	10902.13	133.73
n200cl20co6	1759256	1537680	12.59%	12015.74	126.00
n200cl10co9	1853208	1843480	0.52%	11444.06	246.15
n200cl20co9	2544339	2315700	8.99%	11771.07	167.20
n300cl15co3	1343967	1340220	0.28%	44270.75	341.30
n300cl20co3	1436707	1237820	13.84%	37168.43	262.05
n300cl15co6	2780507	2780310	0.01%	35019.19	503.98
n300cl20co6	3532850	3247430	8.08%	54649.31	309.74
n300cl15co9	4660579	4601520	1.27%	54290.92	485.91
n300cl20co9	4602790	4150960	9.82%	44664.44	645.59

TS did not outperform or get the same result as KS in only one instance, in which case it was only 0.24% worse than the KS solution.

While the percentage improvements seem small, one should note that the objective values for these instances are quite large. Consequently, although there is a significant decrease in reload costs in absolute terms, it is smaller in percentage terms. Although the running times of KS and TS cannot be directly compared as they were run on different computers, one can safely argue that TS runs faster than KS. The running times in the table indicate that TS is obtained on average about 100 times faster than KS; while comparing the speed of the two machines at www.spec.org suggests that there is at most a factor of 4 speedup between the two machines.

6. CONCLUSIONS

We considered the RCSTP and proposed several heuristics for it. These include a construction heuristic to obtain an initial solution, and local search and tabu search heuristics that are based on a tree–nontree edge swap neighborhood. We described how to find the best edge swap efficiently by using preprocessed information. These include demand information, path reload costs, and predecessors for a given spanning tree. We also explain how to efficiently update the data stored after an edge swap is performed. In addition to the RCSTP, we considered two additional variants—the single-source RCSTP where all of the demand originates at the root node, and the SSFRCTSP which has fixed reload costs and the costs are calculated with reference to paths that originate at a given source node. For each of these problems, we considered both unit and nonunit reload cost versions of the problem.

While there are some similarities between the local search algorithm proposed in this paper and the one in Ahuja and Murty [1], there are several *significant differences*. First, Ahuja and Murty [1] do not explore the neighborhood exhaustively. In one iteration, they take a tree edge and consider all possible nontree edge swaps. If the swap between the selected tree edge and the best nontree edge (i.e., the nontree edge that results in the lowest cost) improves the current solution, they implement the swap and do not consider other tree edges. On the other hand, we exhaustively explore the entire neighborhood to find the lowest cost tree–nontree edge swap. Second, because the reload costs are dependent on adjacent edges the update of path costs on the tree for the RCSTP is done slightly differently. Third, our heuristic assumes that demand is asymmetric while they assume the demand is symmetric. Finally, we store and update the $\boldsymbol{Q}^-, \boldsymbol{Q}^+$ matrices which is the key to our fast running times. For a given edge (s, t) to be deleted from the tree (which will then create two subtrees T_1 containing s and T_2 containing t), Ahuja and Murty [1] compute the equivalent of q^-_{it} for $i \in T_1$ and q^-_{is} for $i \in T_2$. They compute this quantity *from scratch* in each iteration. On the other hand, if $\boldsymbol{Q}^-, \boldsymbol{Q}^+$ matrices were stored (i.e., all q values instead of a subset) and updated as described in this paper, it would be possible to (i) explore a larger neighborhood in a single iteration of a local search algorithm for the OCSTP, and (ii) improve the running time from $\mathcal{O}(n^3)$ to $\mathcal{O}(nm)$ for one edge swap iteration in Ahuja and Murty's [1] local search algorithm. We discuss this further in the Appendix.

The quality of our heuristics is validated by our computational experiences on a large data set. On this test set of 630 instances (that includes benchmark instances from [6]), LS improves upon IS in 416 instances by an average of 23.62%, and TS improves upon LS in 364 instances by an average of 35.79%. Out of 495 test instances from this set that we know the optimal solution for IS is optimal 113 times, LS is optimal 224 times, and TS is optimal 481 times. The benefit of tabu search over local search is much clearer on the single-source problems. Of the 420 single-source instances LS is optimal in 151 instances. Of the remaining 269 instances TS improves upon LS 268 times. Comparisons on a second set of benchmark instances from Khalil and Singh [9] tell a similar story. TS improves upon the best known solution in 32 out of 44 instances, and is inferior once. The magnitude of (percentage) improvement of TS on this second data set is larger as the instances get larger.

While we have been able to obtain solutions to large instances rapidly with our heuristics the mathematical programming formulations for the problem (proposed in [6]) are unable to solve instances with 50 nodes in the case of all pairs and more than 100 nodes in the single-source case. This is due to the fact that they are multicommodity flow formulations that rapidly become intractable as the size of the problem increases. One avenue for future research is the development of new formulations and/or exact mathematical programming approaches (e.g., column generation, Benders

decomposition, etc.) for reload cost problems that remain tractable for large-scale reload cost problems.

APPENDIX

We first review the tree–nontree edge swap calculation in Ahuja and Murty [1]. In order to find the best nontree edge to replace a tree edge $e_{ij} \in T$ (suppose the removal of e_{ij} creates two subtrees T_1 containing i and T_2 containing j as described before), the authors first compute the equivalent of q_{si}^- for all $s \in T_1$ and q_{sj}^- for all $s \in T_2$, which takes $\mathcal{O}(n^2)$ time. Note that we do not consider q_{si}^+ here since the demand is assumed to be symmetric and the computations are identical to q_{si}^-. To simplify notation, we drop "–" and "i" from q_{si}^- and simply state it as q_s, which is equal to q_{si}^- if $s \in T_1$ and q_{sj}^- if $s \in T_2$. For a given node $s \in T_1$, Ahuja and Murty calculate

$$h_s = \sum_{t \in T_1} q_t u_{ts} \tag{1}$$

where u_{ts} denotes the path distance from t to s in T. Intuitively, h_s aggregates, for every node $t \in T_1$, the cost of carrying the total demand from node t to the nodes in T_2 (i.e., q_t) to node s. The same calculation is carried out for each node $s \in T_2$ in a similar fashion. After h_s is calculated for all $s \in T$, which takes $\mathcal{O}(n^2)$ time, the cost of carrying the total demand between T_1 and T_2 through a nontree edge e_{kl} is calculated as

$$\alpha_{kl} = h_k + W b_{kl} + h_l$$

where $W = \sum_{s \in T} q_s$ and b_{kl} is the weight (distance) of e_{kl}. The best nontree edge swap for e_{ij} can be identified by calculating α_{kl} for all $e_{kl} \in E \backslash T$ such that $T \backslash \{e_{ij}\} \cup \{e_{kl}\}$ is a spanning tree. Let $\alpha^* = \min_{e \in E \backslash T} \alpha_e$; then $\alpha_{ij} - \alpha^*$ denotes the cost savings achieved by the best nontree edge swap with e_{ij}. In Ahuja and Murty, given a tree edge, q_s and h_s are calculated for all $s \in T$, which takes $\mathcal{O}(n^2)$ time. Then all nontree edges are evaluated, which takes $\mathcal{O}(m)$ time, so finding the nontree edge with the lowest cost takes $\mathcal{O}(n^2) + \mathcal{O}(m) = \mathcal{O}(n^2)$ time.

We now explain how a slight improvement in running time can be obtained using preprocessed information and updating procedures as for the RCSTP. Specifically we show how h_s can be calculated for all $s \in T$ in $\mathcal{O}(n)$ time. For node $s \in T_1$, the removal of edge $e_{s,pred(i,s)}$ partitions T_1 into two subtrees Ω_1 and Ω_2 where $s \in \Omega_1$ and $i, pred(i,s) \in \Omega_2$. Then define $\bar{q}_s = \sum_{t \in \Omega_1} q_t$ as the total demand from the nodes in Ω_1 to nodes in T_2. Let ψ_s be the neighbors of s that are in Ω_1; then $\bar{q}_s$ can be restated as

$$\bar{q}_s = q_s + \sum_{t \in \psi_s} \bar{q}_t.$$

If s is a leaf node, then $\bar{q}_s = q_s$. Therefore, starting from the leaf nodes, $\bar{q}_s$ can be calculated recursively for all $s \in T_1$ using BFS. Since the total number of additions is bounded by the number of edges in T_1, calculating $\bar{q}_s$ for all $s \in T_1$ takes $\mathcal{O}(n)$ time. (In a similar fashion $\bar{q}_s$ can be calculated for all $s \in T_2$ in $\mathcal{O}(n)$ time). Given a tree edge $e_{ij} \in T$, we first compute h_i in $\mathcal{O}(n)$ time as in (1). However, after that we now show how we can recursively calculate h_s for any other node s in T_1 in $\mathcal{O}(1)$ time (by using $\bar{q}_s$ and $\bar{q}_i$). (h_s can be calculated for $s \in T_2$ in a similar fashion provided that h_j is calculated as in (1).)

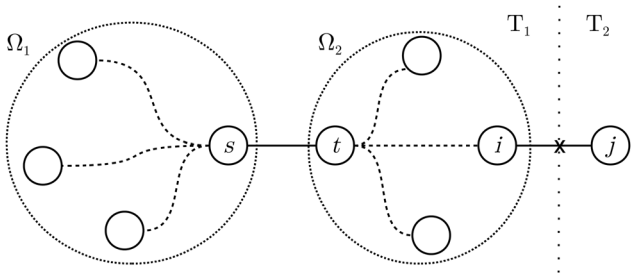

FIG. 7. Removal of edge e_{st} from T_1 creates two subtrees Ω_1 and Ω_2. Given h_t, h_s can be calculated by accounting for the demand that no longer passes from s to t (i.e., $\bar{q}_s$) across edge e_{st}, and the demand that is now carried from t to s (i.e., $\bar{q}_i - \bar{q}_s$) across edge e_{st}.

Consider Figure 7. For node $s \in T_1$, let $t = pred(i,s)$. Given h_t, h_s can be calculated by accounting for the demand that no longer passes from s to t (i.e., $\bar{q}_s$) and the demand that is now carried from t to s (i.e., $\bar{q}_i - \bar{q}_s$). Thus h_s can be stated as

$$h_s = h_t - b_{st}\bar{q}_s + b_{st}(\bar{q}_i - \bar{q}_s) = h_t + b_{st}(\bar{q}_i - 2\bar{q}_s),$$

or equivalently,

$$h_s = h_{pred(i,s)} + b_{s,pred(i,s)}(\bar{q}_i - 2\bar{q}_s),$$

which can be calculated in $\mathcal{O}(1)$ time recursively applying BFS from i.

In summary, $\bar{q}_s$ and h_s for all $s \in T$ can be calculated in $\mathcal{O}(n)$ time via q_s, which is stored and updated after each swap. Then for a given tree edge, finding the nontree edge with the lowest cost becomes $\mathcal{O}(n) + \mathcal{O}(m) = \mathcal{O}(m)$. This results in an overall running time of $\mathcal{O}(nm)$ instead of $\mathcal{O}(n^3)$ for a single local search iteration.

REFERENCES

[1] R. Ahuja and V. Murty, Exact and heuristic algorithms for the optimum communication spanning tree problem, Transp Sci 21 (1987), 163–170.

[2] E. Amaldi, G. Galbiati, and F. Maffioli, On minimum reload cost paths, tours, and flows, Networks 57 (2011), 254–260.

[3] E. Amaldi, L. Liberti, F. Maffioli, and N. Maculan, Edge-swapping algorithms for the minimum fundamental cycle basis problem, Math Methods Oper Res 69 (2009), 205–233.

[4] G. Galbiati, The complexity of a minimum reload cost diameter problem, Discr Appl Math 156 (2008), 3494–3497.

[5] G. Galbiati, S. Gualandi, and F. Maffioli, On minimum reload cost cycle cover, Discr Appl Math 164 (2014), 112–120.

[6] I. Gamvros, L. Gouveia, and S. Raghavan, Reload cost trees and network design, Networks 59 (2012), 365–379.

[7] B. Gfeller, Faster swap edge computation in minimum diameter spanning trees, Algorithmica 62 (2012), 169–191.

[8] L. Gourvès, A. Lyra, C. Martinhon, and J. Monnot, The minimum reload path, trail and walk problems, Discr Appl Math 158 (2010), 1404–1417.

[9] A. Khalil and A. Singh, A swarm intelligence approach to the minimum reload cost spanning tree problem, 1st Int Conf Parallel Distrib Grid Comput, 2010, pp. 260–265. IEEE Piscataway, NJ, USA.

[10] R. Silva, D. Silva, M. Resende, G. Mateus, J. Gonçalves, and P. Festa, An edge-swap heuristic for generating spanning trees with minimum number of branch vertices, Optim Lett 8 (2014), 1225–1243.

[11] H.C. Wirth and J. Steffan, Reload cost problems: minimum diameter spanning tree, Discr Appl Math 113 (2001), 73–85.

[12] B. Wu, C.Y. Hsiao, and K.M. Chao, The swap edges of a multiple-sources routing tree, Algorithmica 50 (2012), 299–311.